PCT WORLD INTELLECTUAL PROPERTY ORGANIZATION
International Bureau

INTERNATIONAL APPLICATION PUBLISHED UNDER THE PATENT COOPERATION TREATY (PCT)

(51) International Patent Classification 5 : G06F 9/44, 9/46

A1

(11) International Publication Number: WO 94/11810

(43) International Publication Date: 26 May 1994 (26.05.94)

(21) International Application Number: PCT/US93/10965

(22) International Filing Date: 12 November 1993 (12.11.93)

(30) Priority data:
07/975,775 13 November 1992 (13.11.92) US

(71) Applicant: MICROSOFT CORPORATION [US/US]; One Microsoft Way, Redmond, WA 98052-6399 (US).

(72) Inventors: ATKINSON, Robert, G. ; 17926 N.E. 196th Street, Woodinville, WA 98072 (US). CORBETT, Tom ; 2085 Oakmont Way, Eugene, OR 97401 (US). JUNG, Edward, K. ; 1518 First Avenue South, Seattle, WA 98134 (US). WILLIAMS, Antony, S. ; 22542 N.E. 46th Street, Redmond, WA 98053 (US).

(74) Agents: LAWRENZ, Steven, D. et al.; Seed and Berry, 6300 Columbia Center, 701 Fifth Avenue, Seattle, WA 98104-7092 (US).

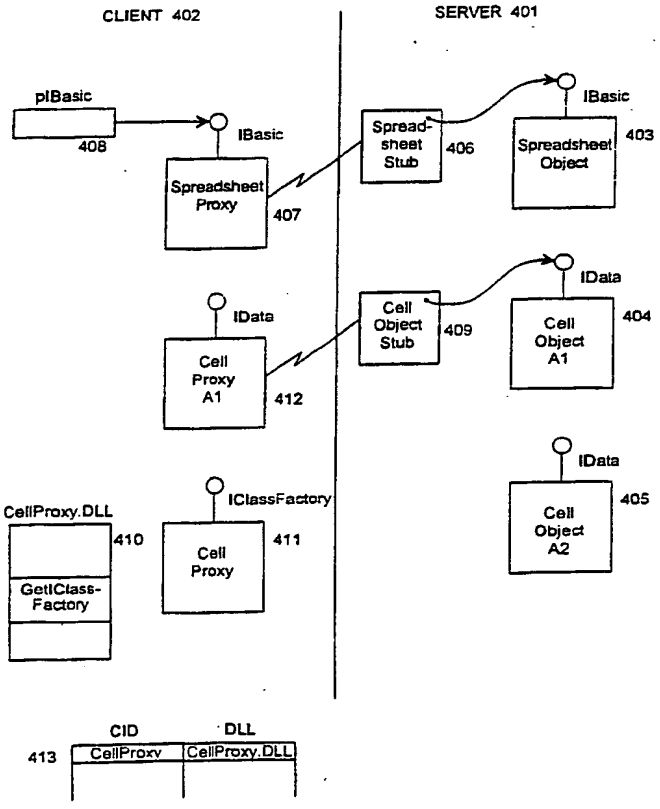
(81) Designated States: CA, JP, European patent (AT, BE, CH, DE, DK, ES, FR, GB, GR, IE, IT, LU, MC, NL, PT, SE).

Published
With international search report.
Before the expiration of the time limit for amending the claims and to be republished in the event of the receipt of amendments.

(54) Title: A METHOD AND SYSTEM FOR MARSHALLING INTERFACE POINTERS FOR REMOTE PROCEDURE CALLS

(57) Abstract

A computer method and system for passing a pointer to an interface from a server process to a client process. In a preferred embodiment, the server process instantiates an object that has multiple interfaces. The server process identifies an interface to pass to the client process and creates a stub object for receiving a request to invoke a function member of the interface and for invoking the requested function member upon receiving the request. The server process then sends an identifier of the stub to the client process. When the client process receives the identifier of the stub, it instantiates a proxy object for receiving requests to invoke a function member of the interface and for sending the request to the identified stub. The client process can then invoke the function members of the interface by invoking function members of the proxy object. The proxy object sends a request to the identified stub. The identified stub then invokes the corresponding function member of the interface.



Description

A METHOD AND SYSTEM FOR MARSHALLING INTERFACE POINTERS FOR REMOTE PROCEDURE CALLS

Technical Field

This invention relates generally to a computer method and system for passing data and, more specifically, to a method and system for passing pointers to objects between processes.

Background of the Invention

Computer systems typically have operating systems that support multitasking. A multitasking operating system allows multiple tasks (processes) to be executing concurrently. For example, a database server process may execute concurrently with many client processes, which request services of the database server process. A client process (client) may request services by issuing a remote procedure call (RPC). A remote procedure call allows a server process (server) to invoke a server procedure on behalf of the client. To issue a remote procedure call, the client packages the procedure name and the actual in-parameters for the procedure into an interprocess communications message and sends the message to the server. The server receives the message, unpackages the procedure name and any actual in-parameters, and invokes the named procedure, passing it the unpackaged in-parameters. When the procedure completes, the server packages any out-parameters into a message and sends the message to the client. The client receives the message and unpackages the out-parameters. The process of packaging parameters is known as marshalling, and the process of unpackaging parameters is known as unmarshalling.

Parameters may be marshalled by storing a copy of the value of the actual parameter in a message. For certain types of parameters, the marshalling may involve more than simply storing a copy of the value. For example, a floating point value may need to be converted from the format of one computer system to the format of another computer system when the processes reside on different computer systems.

The copying of the value of an actual parameter has a couple disadvantages. First, when a copy is passed changes to the original value are not reflected in the copy. For example, if a parameter representing a time of day value is passed from a client to a server by copying, then the copy that the server receives is not updated as the client updates its time of day value. Second, with

type are stored in a compound document.) For example, a word processing document that contains a spreadsheet object generated by a spreadsheet program is a compound document. A word processing program allows a user to embed a spreadsheet object (e.g., a cell) within a word processing document. To allow this embedding, the word processing program would be compiled using the class definition of the object to be embedded to access function members of the embedded object. Thus, the word processing program would need to be compiled using the class definition of each class of objects that can be embedded in a word processing document. To embed an object of a new class into a word processing document, the word processing program would need to be recompiled with the new class definition. Thus, only objects of classes selected by the developer of the word processing program can be embedded. Furthermore, new classes can only be supported with a new release of the word processing program.

To allow objects of an arbitrary class to be embedded into compound documents, interfaces (abstract classes) are defined through which an object can be accessed without the need for the word processing program to have access to the class definitions at compile time. An abstract class is a class in which a virtual function member has no implementation (pure). An interface is an abstract class with no data members and whose virtual functions are all pure.

The following class definition is an example definition of an interface. In this example, for simplicity of explanation, rather than allowing any class of object to be embedded in its documents, a word processing program allows spreadsheet objects to be embedded. Any spreadsheet object that provides this interface can be embedded, regardless of how the object is implemented. Moreover, any spreadsheet object, whether implemented before or after the word processing program is compiled, can be embedded.

```
class   ISpreadSheet
        { virtual void File() = 0;
          virtual void Edit() = 0;
          virtual void Formula() = 0;
          virtual void Format() = 0;
          virtual void GetCell (string RC, cell *pCell) = 0;
          virtual void Data() = 0;
          ...
        }
```

Each spreadsheet developer would implement the IBasic interface and, optionally, the IDatabase interface.

At run time, the word processing program would need to determine whether a spreadsheet object to be embedded supports the IDatabase interface. To make this determination, another interface is defined (that every spreadsheet object implements) with a function member that indicates which interfaces are implemented for the object. This interface is known as IUnknown and is defined by the following.

```
class IUnknown
    { virtual boolean QueryInterface (iidInterface, pInterface) = 0;
    ...
    }
```

The IUnknown interface defines the function member (method) QueryInterface. The method QueryInterface is passed an interface identifier (e.g., "IDatabase") and returns a pointer to the implementation of the identified interface for the object for which the method is invoked. If the object does not support the interface, then the method returns a false.

The IDatabase interface and IBasic interface inherit the IUnknown interface. Inheritance is well known in object-oriented techniques by which a class definition can incorporate the data and function members of previously-defined classes. The following definitions illustrate the use of the IUnknown interface.

```
class IDatabase : IUnknown
    { virtual void Data() = 0;
    }

class IBasic : IUnknown
    { virtual void File() = 0;
      virtual void Edit() = 0;
      virtual void Formula() = 0;
      virtual void Format() = 0;
      virtual void GetCell (string RC, cell *pCell) = 0;
    ...
    }
```

The method CreateInstance instantiates an object and returns a pointer pInterface to the interface of the object designated by argument iidInterface.

Although the use of the above described interfaces can be used to facilitate embedding objects in a compound document, an efficient technique is needed for allowing pointers to objects (interfaces) to be passed as parameters in a remote procedure call. The passing of pointers avoids the overhead of copying objects and allows the receiving process to see changes that the sending process makes to the object.

Summary of the Invention

It is an object of the present invention to provide a method and system for allowing a client process to access an interface of an object instantiated in a server process.

It is another object of the present invention to provide a method and system for allowing an object to implement methods for class-specific (custom) marshalling and unmarshalling of pointers to the object.

It is another object of the present invention to provide a method and system for passing pointers to interfaces of object between processes.

These and other objects, which will become apparent as the invention is more fully described below, are obtained by a method and system for passing a pointer to an interface from a server process to a client process. In a preferred embodiment, the server process instantiates an object that has multiple interfaces. The server process identifies an interface to pass to the client process and creates a stub object for receiving a request to invoke a function member of the interface and for invoking the requested function member upon receiving the request. The server process then sends an identifier of the stub to the client process. When the client process receives the identifier of the stub, it instantiates a proxy object for receiving requests to invoke a function member of the interface and for sending the request to the identified stub. In another embodiment, the server process sends to the client process the class of the proxy object and the client dynamically loads code to instantiate the proxy object. In another embodiment, objects implement an interface with methods for custom marshalling and unmarshalling pointers to interfaces of the object. These methods are invoked to marshal and unmarshal pointers to interfaces.

an interprocess communications message address for the stub, storing a pointer to an interface of object 301 within the stub, and packaging the message address of the stub and a class identifier of the proxy into a message. The server then sends the message to the client. When the client receives the message, the client unmarshals the pointer to an interface of object 301 by retrieving the class identifier of the proxy and the stub message address, dynamically loading code to create an instance of the class identifier class identified by the retrieved instantiating proxy 303, and storing the stub message address with the proxy 303. The client then accesses the interface of object 301 through proxy 303.

Proxy 303 is an object that implements the same interface as the interface of object 301, but with a different implementation. Each method of proxy 303 marshals its name and its actual parameters into a message, sends the message to stub 302, waits for a return message from stub 302, and unmarshals any returned parameters. Table 1 lists sample pseudocode for a proxy method named "File."

Table 1

```
void File (string Name, int OpenMode, int Status)
        {package "File" into message
         package Name into message
         package OpenMode into message
         send message to stub
         wait for message from stub
         unpackage Status from message
        }
```

Stub 302 is an object that implements an interface that receives messages from the client, unmarshals the method name and any in-parameters, invokes the named method of object 301, marshals any out-parameters into a message, and sends the message to the proxy 303. Table 2 lists sample pseudocode for a stub.

Figure 4B is a block diagram illustrating the marshalling of a cell object to the client. When a client wants to retrieve the formula of cell A1 represented as cell object 404, the client executes the following statements.

```
pIBasic->GetCell("A1", pCell);
formula = pCell -> GetFormula();
```

The spreadsheet proxy 407 is pointed to by pointer pIBasic 408. The client first invokes the spreadsheet proxy method GetCell. The method GetCell packages the method name "GetCell" and the string "A1" into a message and sends the message to spreadsheet stub 406. The spreadsheet stub 406 unpackages the method name and string. The spreadsheet stub 406 then invokes the GetCell method of the spreadsheet object 403. The method GetCell returns to the spreadsheet stub 406 a pointer to cell object 404. The spreadsheet stub 406 then marshals the cell pointer by creating cell stub 409 for cell object 404, assigning a message address to cell stub 409, packaging the message address and an unmarshal class identifier (described below) into a message, and sending the message to the spreadsheet proxy 407. When the spreadsheet proxy 407 receives the message, method GetCell then unmarshals the pointer to the cell object 404.

Figure 4C is a block diagram illustrating the data structures used during the unmarshalling of a cell pointer. The method GetCell unmarshals the pointer to cell A1 by first retrieving the message address and the unmarshal class identifier ("CellProxy") from the received message. In a preferred embodiment, the persistent registry 413 contains the name of a dynamic link library for each class. Dynamic link libraries are libraries of routines (methods, functions, etc.) that are loaded at run time of a program. Dynamic link libraries are described in the reference "Programming Windows," by Charles Petzold and published by Microsoft Press. Each dynamic link library for a class contains a function GetIClassFactory which returns a pointer to an IClassFactory interface for the class. The method GetCell loads the dynamic link library 410 for the retrieved unmarshal class identifier (the unmarshal class) and invokes the function GetIClassFactory which returns a pointer to the IClassFactory interface 411. The method GetCell then invokes the method CreateInstance of the IClassFactory interface 411 to create cell proxy 412. The cell proxy 412 is then initialized with the retrieved message address for cell stub 409. The method GetCell then returns with the pointer to the cell proxy 412. The client can then access the cell object 404 through the cell proxy 412.

(step 505) and MarshalInterface (step 511) preferably determine if the object implements the IMarshal interface and invokes the function members of the IMarshal interface as appropriate.

Figures 6 through 10 are flow diagrams of the methods and functions invoked during the marshalling and unmarshalling of a pointer to a cell object. As described below, these methods and functions implement standard marshalling.

Figure 6 is a flow diagram of the function MarshalInterface. The function has the following prototype.

void MarshalInterface (pstm, iid, pInterface, DestContext)

This function marshals the designated pointer (pInterface) to an interface for an object into the designated message (pstm). In step 601, the function determines whether the object implements custom marshalling. The function invokes the method QueryInterface of the interface to retrieve a pointer to an IMarshal interface. If the object implements custom marshalling, then a pointer (pMarshal) to the IMarshal interface for the object is returned and the function continues at step 603, else the function continues at step 602. In step 602, the function invokes the function GetStandardMarshal to retrieve a pointer (pMarshal) to an IMarshal interface with default marshalling methods. In step 603, the function invokes the method IMarshal::GetUnmarshalClass pointed to by the retrieved pointer. The method GetUnmarshalClass returns the class identifier of the class that should be used in the unmarshalling process to instantiate a proxy for the designated interface (iid). In step 604, the function invokes the function Marshal to marshal the unmarshal class identifier to the designated message. In step 605, the function invokes the method IMarshal::MarshalInterface pointed to by the retrieved pointer (pMarshal). The method MarshalInterface marshals the designated interface pointer (pInterface) to the designated message. The method then returns.

Figure 7 is a flow diagram of a sample standard implementation of the method MarshalInterface. The method has the following prototype.

void IMarshal::MarshalInterface (pstm, iid, pInterface, DestContext)

The method MarshalInterface marshals the designated pointer to an interface (pInterface) to the designated message (pstm). In step 701, the method invokes function MakeStub. The function MakeStub makes a stub for the designated

The method UnMarshalInterface initializes the newly created proxy and returns a pointer (pInterface) to the designated interface (iid). In step 1001, the method invokes function UnMarshal to unmarshal the stub message address from the designated message. In step 1002, the method stores the stub message address. In step 1003, the method retrieves a pointer to the designated interface. The method UnMarshalInterface returns.

Figure 11 is a flow diagram of the function GetClassObject. The function has the following prototype.

```
void GetClassObject (cid, iid, pInterface)
```

The function GetClassObject returns a pointer (pInterface) to the designated interface (iid) of the designated class (cid). In step 1101, the function retrieves the name of the dynamic link library for the designated class from the persistent registry. In step 1102, the function loads the dynamic link library. In step 1103, if the designated interface is the IClassFactory, then the function continues at step 1104, else the function returns. In step 1104, the function invokes the function GetIClassFactory to get the pointer to the IClassFactory interface. The function GetIClassFactory is preferably provided by the developer of the object. The function GetClassObject then returns.

A developer of an object can provide an implementation of the IMarshal interface to provide custom marshalling and unmarshalling for the object. The IMarshal interface is defined in the following.

```
class IMarshal: IUnknown
        { virtual void GetUnmarshalClass (iid, pInterface, DestContext, cid) = 0;
          virtual void MarshalInterface(pstm, iid, pInterface, DestContext) = 0;
          virtual void UnMarshalInterface(pstm, iid, pInterface) = 0;
        }
```

A developer may implement custom marshalling to optimize the marshalling process. Figures 12A and 12B are block diagrams illustrating custom marshalling of interface pointers. Figure 12A shows the effects of standard marshalling. In Figure 12A, cell object 1201 in process P1 has been marshalled to process P2 as represented by cell stub 1202 and cell proxy 1203. Cell proxy 1203 has been marshalled to process P3 as represented by cell stub 1204 and cell proxy 1205. Cell proxy 1203 was marshalled using the standard marshalling of the type

pointer to cell object 1301 is marshalled and sent from process P1 to process P2. The custom methods of the IMarshal interface for a cell object are implemented by the following pseudocode.

```
void IMarshal::MarshalInterface (pstm, iid, pInterface, DestContext)
        { if (DestContext = = Shared) then
                { Marshal (pstm, address of data);}
          else
                { MakeStub ("Cell", iid, pInterface, &MessageAddress);
                  Marshal (pstm, MessageAddress)};
        }

void IMarshal::GetUnmarshalClass (iid, pInterface, DestContext, cid)
        { if (DestContext = = Shared)
                {cid = "Cell"}
          else
                {cid = "CellProxy"}
        }

void IMarshal::UnMarshalInterface (pstm, iid, pInterface);
        { UnMarshal (pstm, address of data);
          initialize object to point to shared data
        }
```

The parameter DestContext of the method IMarshal::MarshalInterface indicates whether the data of the cell object is stored in shared memory. If the data is not stored in shared memory, then the equivalent of the standard marshalling is performed. If, however, the data is stored in shared memory, then the address of the shared data is marshalled into the message. The method IMarshal::GetUnmarshalClass determines the unmarshal class based on whether the data is in shared memory. If the data is not in shared memory, then the CellProxy class is the unmarshal class. If the data is in shared memory, then the Cell class is the unmarshal class. The unmarshal class and address of the shared data are sent to process P2. After process P2 instantiates the cell object 1303, the process P2 invokes the custom method IMarshal::UnMarshalInterface, which unmarshals the address of the shared data and initializes the object to point to the data.

Claims

1. A method in a computer system for passing a pointer to an interface of an object from a server process to a client process, the method comprising the steps of:

within the server process,

instantiating the object, the object having a plurality of interfaces, each interface having a function member;

identifying an interface of the object to pass to the client process;

creating a stub for receiving a request to invoke a function member of the interface and for invoking the requested function member upon receiving the request, the stub having an identifier; and

sending the identifier of the stub to the client process; and

within the client process,

receiving the identifier of the stub from the server process; and

instantiating a proxy object for receiving requests to invoke a function member of the interface and for sending the request to the identified stub.

2. The method of claim 1 including the steps of:

within the server process,

sending an unmarshal class identifier identifying the class of the proxy object to instantiate in the instantiating step to the client process; and

within the client process,

receiving the unmarshal class identifier from the server process; and

dynamically loading code to instantiate an object of the class identified by the unmarshal class identifier wherein the step of instantiating a proxy object executes the dynamically-loaded code.

3. The method of claim 2 including the step of:

within the client process,

retrieving a persistent registry entry indicator of the code to dynamically load from a persistent registry, the persistent registry having an entry associating the unmarshal class identifier with the persistent registry entry indicator.

creating a stub for receiving a request to invoke a function member of the object and for invoking the requested function member upon receiving the request, the stub having an identifier; and

sending an unmarshal class identifier identifying the class of a proxy object to instantiate within the client process and the identifier of the stub to the client process; and

within the client process,

receiving the unmarshal class identifier and the identifier of the stub from the server process;

dynamically loading code to instantiate the proxy object of the class identified by the unmarshal class identifier; and

executing the loaded code to instantiate a proxy object of the class identified by the unmarshal class identifier and to initialize the proxy object so that when the proxy object receives requests to invoke a function member of the object, the proxy object sends the request to the identified stub.

8. The method of claim 7 including the step of:

within the client process,

retrieving a persistent registry entry indicator of the code to dynamically load from a persistent registry, the persistent registry having an entry associating the unmarshal class identifier with the persistent registry entry indicator.

9. A method in a computer system of passing a pointer to an object from a server process to a client process, the method comprising the steps of:

within the server process,

instantiating the object having a marshalling function member for performing custom marshalling for a pointer to the object;

invoking the marshalling function member of the object; and

sending to the client process an unmarshal class identifier identifying the class of a proxy object to instantiate in the client process; and

within the client process,

receiving the unmarshal class identifier from the server process;

dynamically loading code to instantiate the object of the class identified by the unmarshal class identifier;

executing the loaded code to instantiate an object of the class identified by the unmarshal class identifier, the object of the class identified by the unmarshal class identifier having an unmarshalling function member; and

receiving the unmarshal class identifier and the identifier of the stub from the first client process;

dynamically loading code to instantiate a second proxy object of the class identified by the unmarshal class identifier; and

executing the loaded code to instantiate a second a proxy object of the class identified by the unmarshal class identifier and to initialize the second proxy object so that when the second proxy object receives requests to invoke a function member of the object, the second proxy object sends the request to the identified stub.

12. The method of claim 11 wherein the object has a plurality of interfaces and the pointer to the object is a pointer to one of the interfaces of the object.

13. A method in a computer system of passing a pointer to an object from a server process to a client process, the method comprising the steps of:

within the server process,

instantiating the object in shared memory that is accessible by the server process and the client process and that has a marshalling function member to store a pointer to data members of the object in a message;

storing an unmarshal class identifier in a message;

invoking the marshalling function member of the object; and

sending the message to the client process; and

within the client process,

receiving the message from the server process;

retrieving the unmarshal class identifier from the message;

dynamically loading code to instantiate an object of the class identified by the unmarshal class identifier;

executing the loaded code to instantiate an object of the class identified by the unmarshal class identifier, the object of the class identified by the unmarshal class identifier having an unmarshalling function member; and

invoking the unmarshalling function member of the object of the class identified by the unmarshal class identifier wherein the unmarshalling function member stores the pointer to the data members in the object so that function members of the object can access the data members in shared memory.

14. The method of claim 13 wherein the unmarshal class identifier identifies the same class as the object instantiated by the server process.

a method for retrieving an unmarshal class identifier;
a marshalling method for marshalling a pointer to an object; and
an unmarshalling method for unmarshalling the pointer to the object.

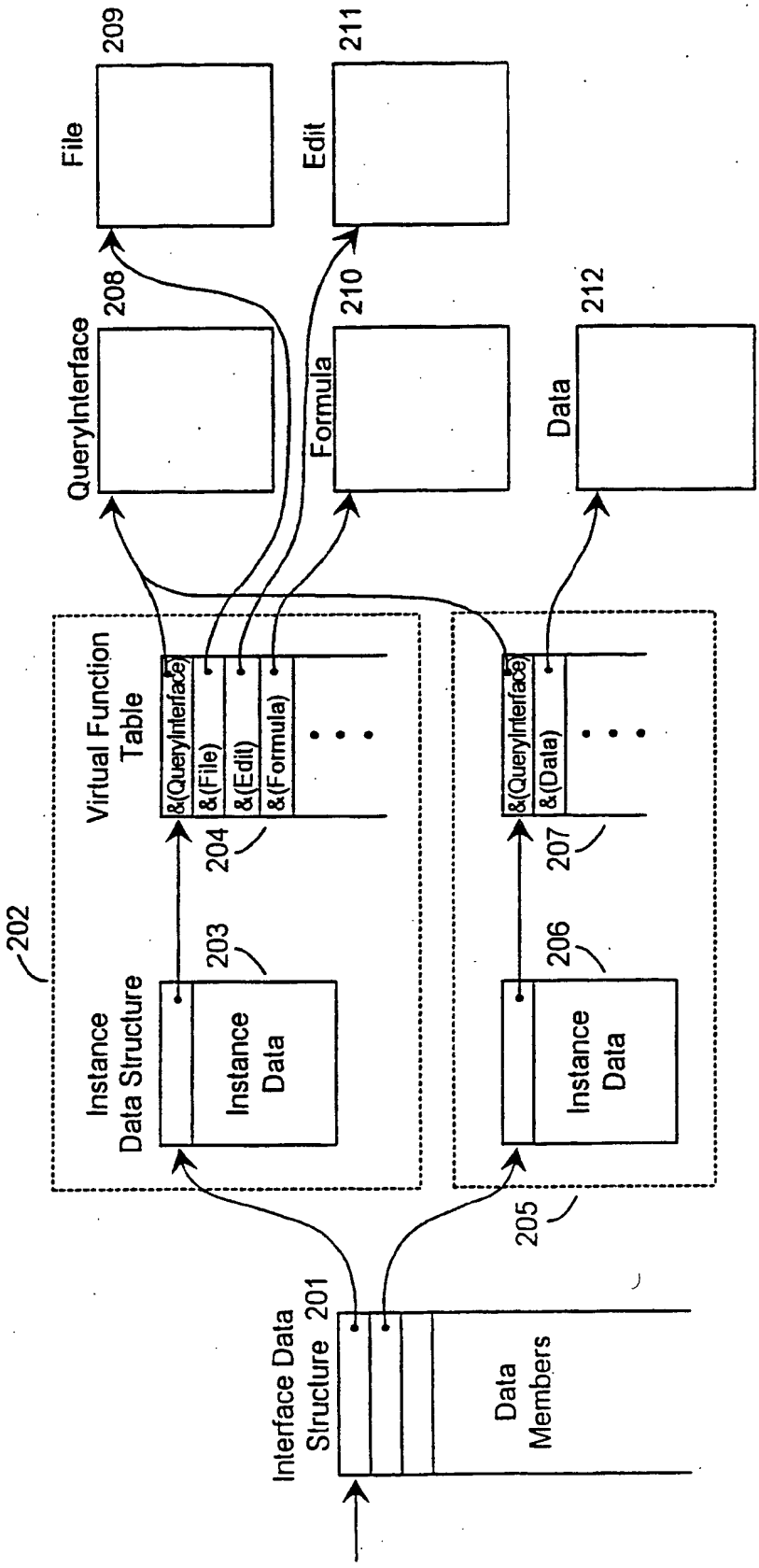


Figure 2

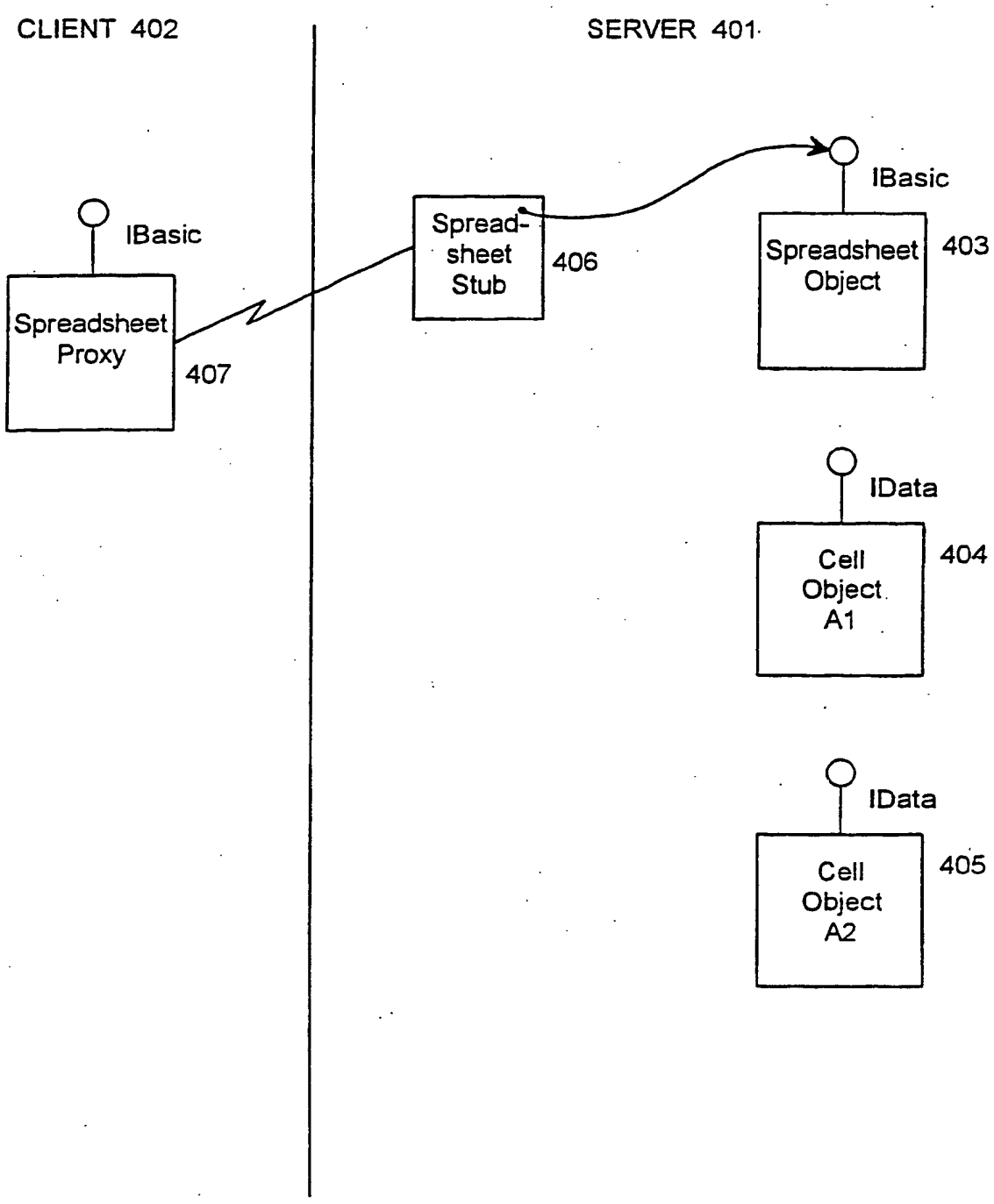

CLIENT 402
SERVER 401
IBasic
Spreadsheet Proxy
407
Spread-sheet Stub
406
IBasic
Spreadsheet Object
403
IData
Cell Object A1
404
IData
Cell Object A2
405

Figure 4A

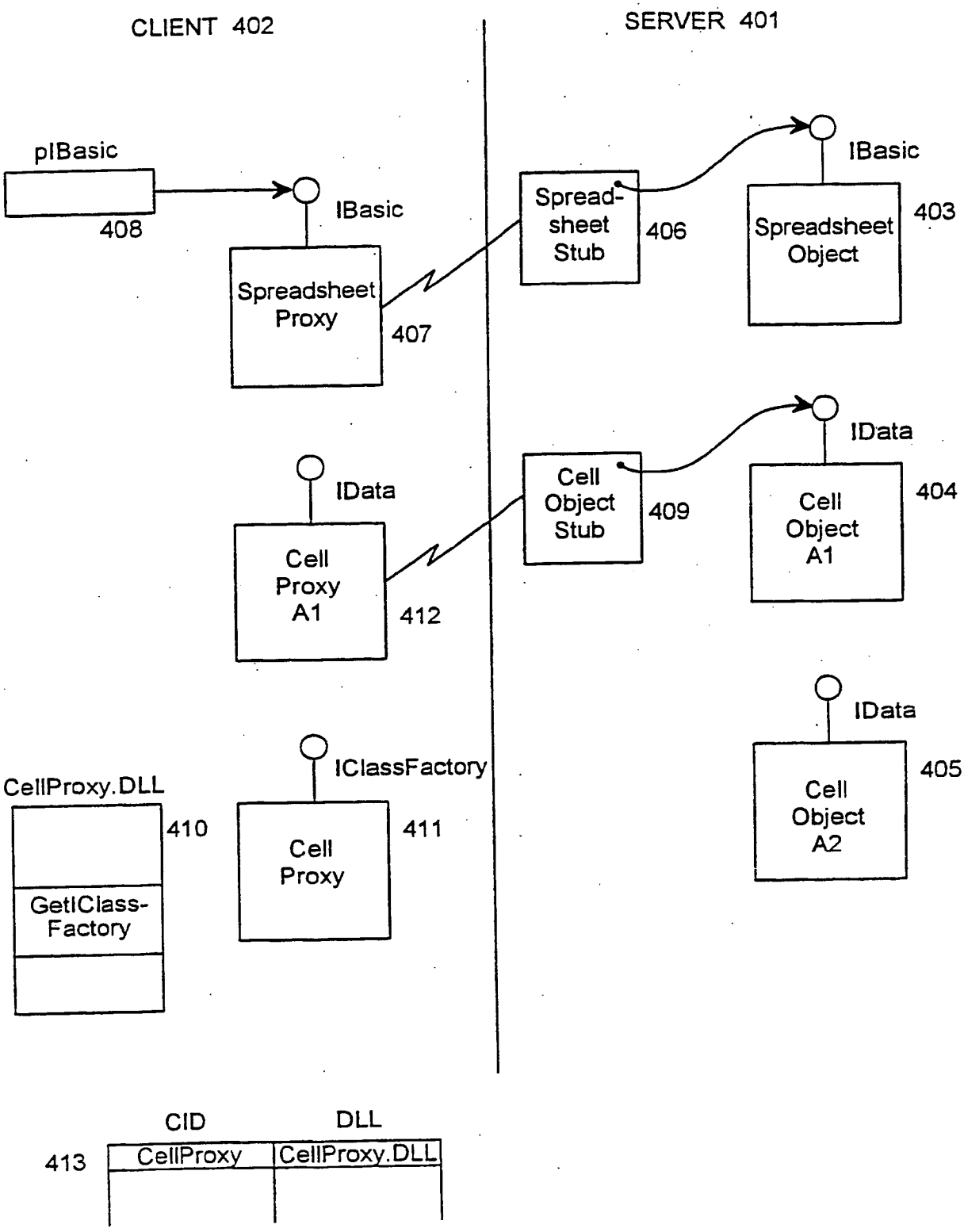

CLIENT 402
SERVER 401
pIBasic
408
IBasic
Spreadsheet Proxy
407
Spread-sheet Stub
406
IBasic
Spreadsheet Object
403
IData
Cell Proxy A1
412
Cell Object Stub
409
IData
Cell Object A1
404
IClassFactory
Cell Proxy
411
IData
Cell Object A2
405
CellProxy.DLL
410
GetIClass-Factory
CID
DLL
413
CellProxy
CellProxy.DLL

Figure 4C

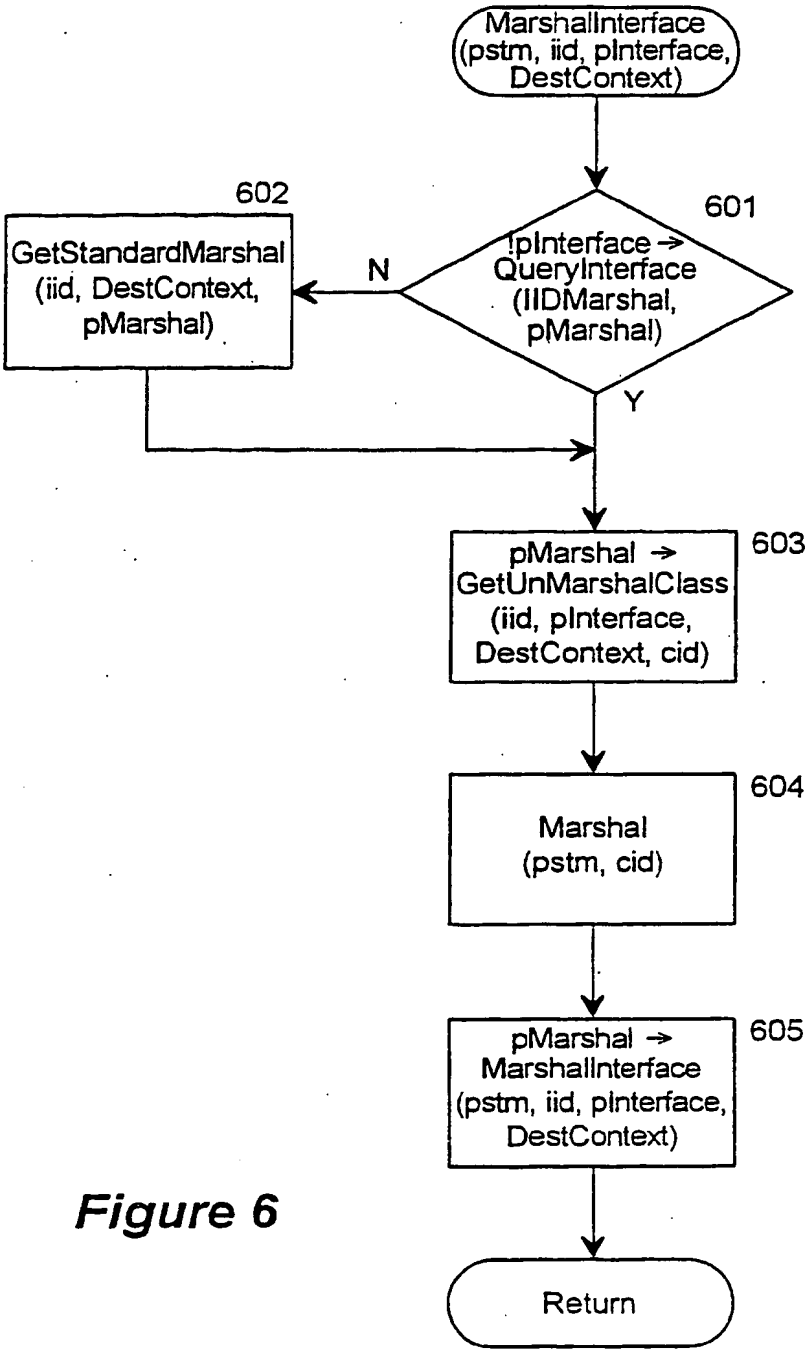

MarshalInterface (pstm, iid, pInterface, DestContext)
601
!pInterface → QueryInterface (IIDMarshal, pMarshal)
N
602
GetStandardMarshal (iid, DestContext, pMarshal)
Y
pMarshal → GetUnMarshalClass (iid, pInterface, DestContext, cid)
603
Marshal (pstm, cid)
604
pMarshal → MarshalInterface (pstm, iid, pInterface, DestContext)
605
Return

Figure 6

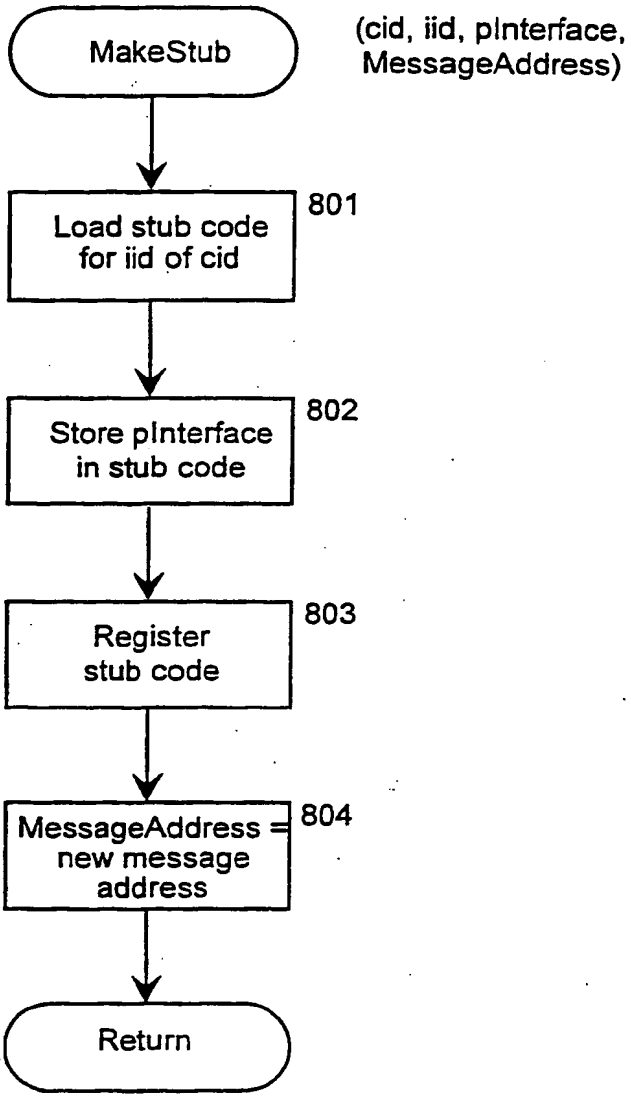

MakeStub
(cid, iid, pInterface, MessageAddress)
Load stub code for iid of cid
801
Store pInterface in stub code
802
Register stub code
803
MessageAddress = new message address
804
Return

Figure 8

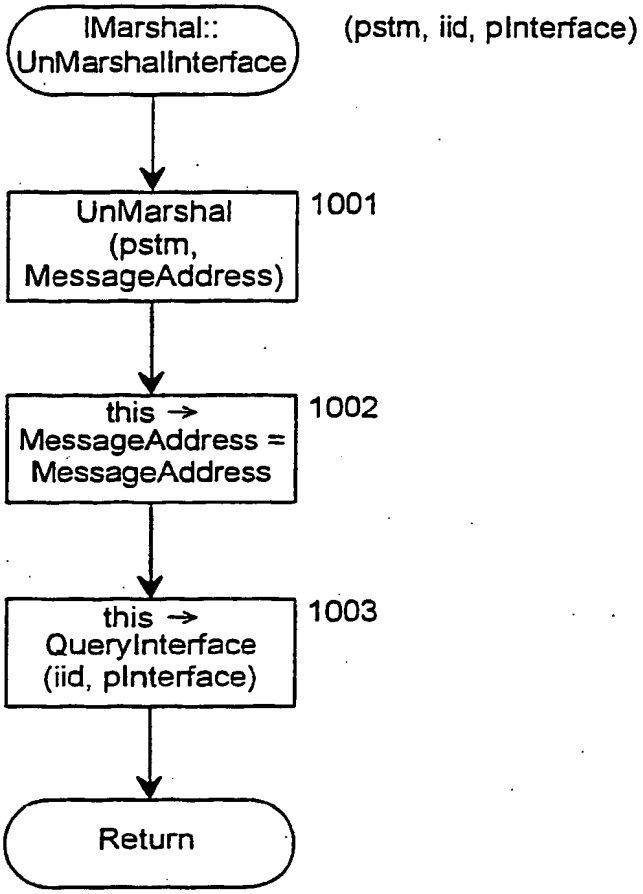

IMarshal::
UnMarshalInterface
(pstm, iid, pInterface)
UnMarshal
(pstm,
MessageAddress)
1001
this →
MessageAddress =
MessageAddress
1002
this →
QueryInterface
(iid, pInterface)
1003
Return

Figure 10

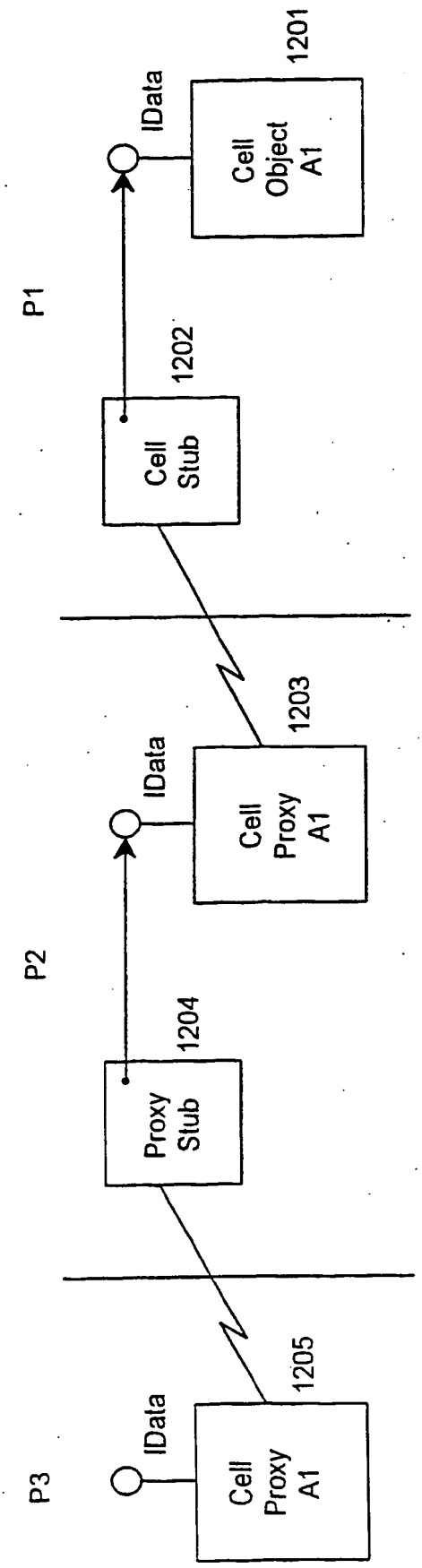


Figure 12A

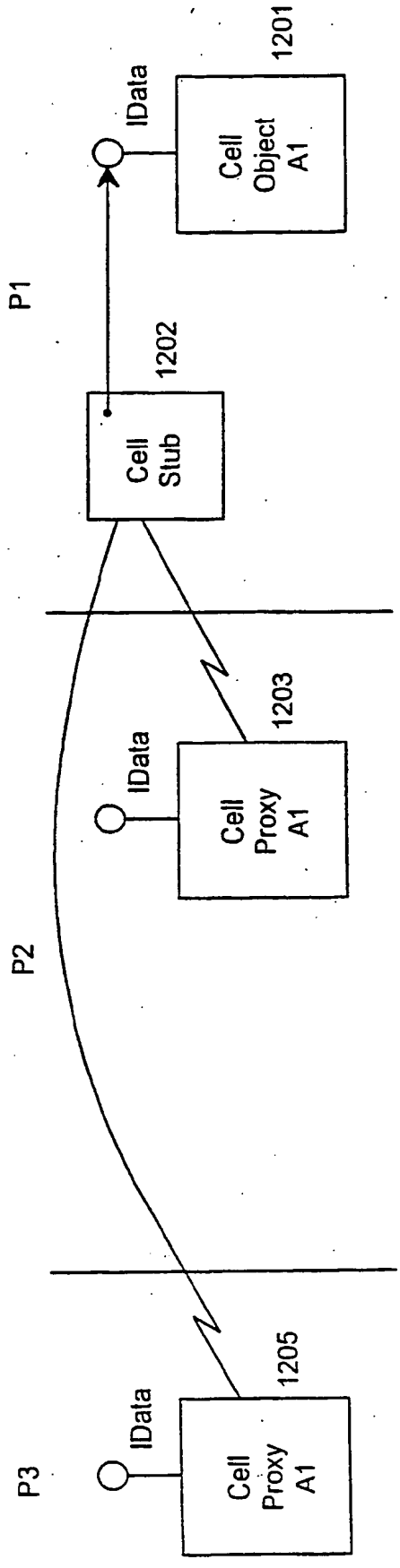


Figure 12B

INTERNATIONAL SEARCH REPORT

Interr ıal Application No
PCT/US 93/10965

A. CLASSIFICATION OF SUBJECT MATTER
IPC 5 G06F9/44 G06F9/46

According to International Patent Classification (IPC) or to both national classification and IPC

B. FIELDS SEARCHED

Minimum documentation searched (classification system followed by classification symbols)
IPC 5 G06F

Documentation searched other than minimum documentation to the extent that such documents are included in the fields searched

Electronic data base consulted during the international search (name of data base and, where practical, search terms used)

C. DOCUMENTS CONSIDERED TO BE RELEVANT

Category *	Citation of document, with indication, where appropriate, of the relevant passages	Relevant to claim No.
X	6TH INT. CONF. ON DISTRIBUTED COMPUTING SYSTEMS 19 May 1986 , IEEE, WASHINGTON, US pages 198 - 204 MARC SHAPIRO 'Structure and encapsulation in distributed systems: the proxy principle' see page 199, right column, line 23 - page 201, left column, line 53 see page 202, right column, line 22 - page 203, left column, line 30 see page 202, right column, line 34 - page 203, left column, line 11	1,19
Y		2-10, 13-18
A		11,12

	-/--	

[X] Further documents are listed in the continuation of box C.

[X] Patent family members are listed in annex.

* Special categories of cited documents :

"A" document defining the general state of the art which is not considered to be of particular relevance

"E" earlier document but published on or after the international filing date

"L" document which may throw doubts on priority claim(s) or which is cited to establish the publication date of another citation or other special reason (as specified)

"O" document referring to an oral disclosure, use, exhibition or other means

"P" document published prior to the international filing date but later than the priority date claimed

"T" later document published after the international filing date or priority date and not in conflict with the application but cited to understand the principle or theory underlying the invention

"X" document of particular relevance; the claimed invention cannot be considered novel or cannot be considered to involve an inventive step when the document is taken alone

"Y" document of particular relevance; the claimed invention cannot be considered to involve an inventive step when the document is combined with one or more other such documents, such combination being obvious to a person skilled in the art.

"&" document member of the same patent family

Date of the actual completion of the international search	Date of mailing of the international search report
7 April 1994	18.04.94

Name and mailing address of the ISA	Authorized officer
European Patent Office, P.B. 5818 Patentlaan 2 NL - 2280 HV Rijswijk Tel. (+31-70) 340-2040, Tx. 31 651 epo nl, Fax (+31-70) 340-3016	Kingma, Y

Form PCT/ISA/210 (second sheet) (July 1992)

INTERNATIONAL SEARCH REPORT

...formation on patent family members

Interr ...al Application No
PCT/US 93/10965

Patent document cited in search report	Publication date	Patent family member(s)	Publication date
WO-A-9306548	01-04-93	DE-A- 4131380	25-03-93

Form PCT/ISA/210 (patent family annex) (July 1992)